Problem Solving with Genetic Algorithms

Edward P K Tsang
Department of Computer Science
University of Essex
Colchester CO4 3SQ
tel: 01206 872774
email: edward@essex
http://cswww.essex.ac.uk/CSP/edward/edward.html

I. Inspiration from nature

Nature is full of inspirations for human beings. This article describes a class of computer algorithms called the *Genetic Algorithms* (GAs), which are inspired by evolution.

Many problems are intractable for the methods developed so far. One well known example is the *Travelling Salesman Problem* (TSP): given a set of cities and the distance between each pair of them, the problem is to find a tour which visits all the cities with the minimal distance to be travelled. A problem with 10 cities may not be too difficult to deal with. But what about problems with 50, 100 or 500 cities? For large problems, it is often intractable for conventional algorithms to find optimal solutions. To tackle these problems, techniques have been developed to find near-optimal solutions within an acceptable period of time. GA is one such class of techniques.

The idea of GAs comes from natural selection. Nature is very effective in selecting the best species. Species adapt to the environment, and pass their features to their offspring via *genes*. Is it possible for us to learn from nature, and develop an "*environment*" which, like nature, allows near-optimal or optimal solutions to be "*evolved*"? Furthermore, can programs be "*cultivated*", instead of being painstakingly written? Apart from being interesting, could this idea be developed into robust, efficient and effective tools for problem solving? In 1975, John Holland proposed that at least for some problems, this is possible. The idea was not met by great enthusiasm in the first ten years of its introduction, but research in this line has received more and more attention in recent years.

The proposal is to encode solution candidates as strings of building blocks. A string is analogous to a *chromosome* and a building block is analogous to a gene in molecular biology. In nature, each gene may take one of several possible values, called *alleles*. In GAs, binary values (0's and 1's) are the most commonly used, though sometimes multiple values are used. Each string is evaluated and assigned a numerical value which is called the *fitness* of this string. For a GA to work, the fitness of a string must be dependent solely on the values that its building blocks take. In optimization problems, the fitness is normally the function which is to be optimized. For example, in the travelling salesman problem, each string may represent a tour which visits all the cities, and the fitness of it may be the negation of the total distance to be travelled in the tour. In that case, the goal is to find a string which has the maximum fitness. Figure 1 summarizes the terminologies introduced so far. The problem is then about how to manipulate such strings in a way which mimics evolution, so as to "cultivate" strings of high fitness.

II. The Genetic Algorithms

The basic algorithm works as follows. A set of strings, which is called a *population*, is maintained. The population evolves by allowing pairs of members to combine and transform

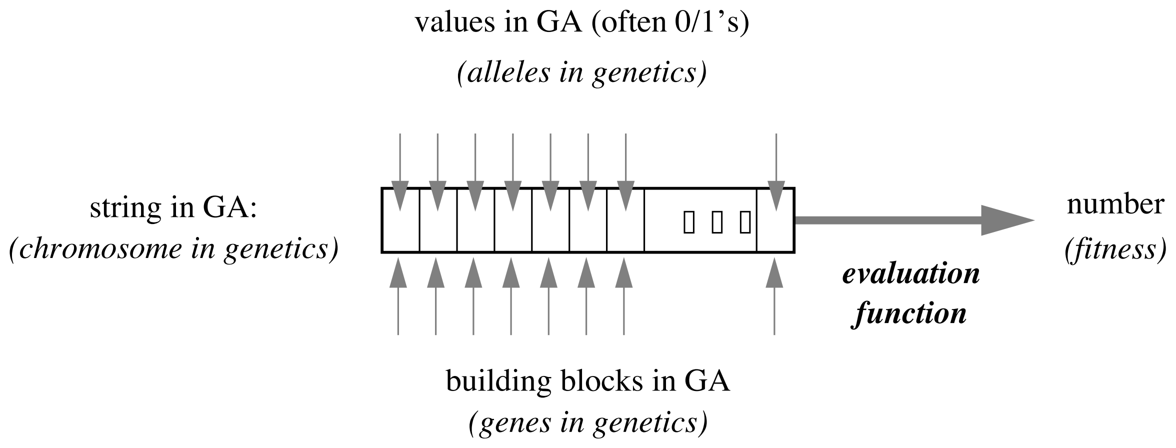


Figure 1 — Representation of candidate solutions in GA:

in order to generate new offspring. This is often called *mating* and *reproduction*. The fitter a string is, the more chance it is given to mate and to reproduce. Therefore, patterns of building blocks which appear in fitter strings will get better chances of being retained. Patterns of building blocks which appear in weaker strings are in greater danger of being eliminated.

To help the readers understand the algorithms, let us first look at one possible way of combining strings. It is observed that in nature, chromosomes exchange part of their genes during reproduction. This is mimicked by a simple artificial combination operation, which is called *crossover*: given a pair of parent strings, an arbitrary cutoff point is picked. Then the two parents exchange their building blocks at the cutoff point. An example of crossover is shown in figure 2. There the parent strings are 1000110 and 0011011. The cutoff point is in this example is between the 4th and 5th building blocks. The first offspring, 1000011, is produced by taking the first 4 blocks of parent 1, and the last three blocks of parent 2. The second offspring is produced by taking the remaining building blocks.

It is observed that in nature, some genes mutate occasionally. This is mimicked by changing the values of building blocks occasionally in GAs. Figure 3 shows a string being mutated. A random block position is picked, and a random value (in this case, a different value) is assigned to it. We have used binary values in our examples, but in general, there could be more possible values for each building block.

The basic control strategy of GAs is summarized in figure 4. To start, an initial population of strings is generated. For the time being, we can imagine the members of this population

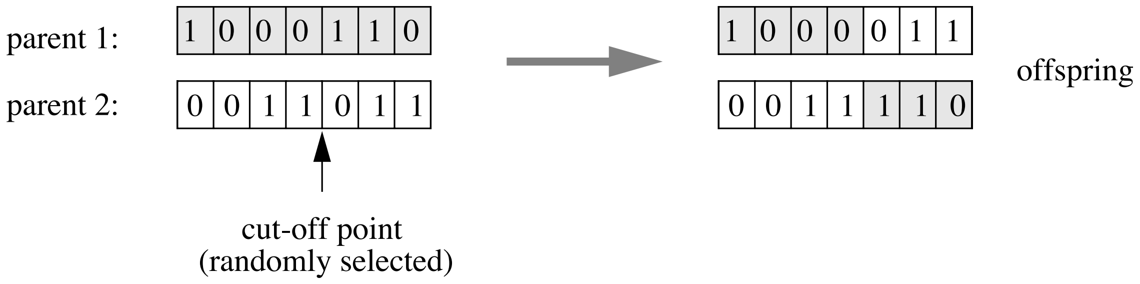


Figure 2 — The Crossover (strings combination) Operator

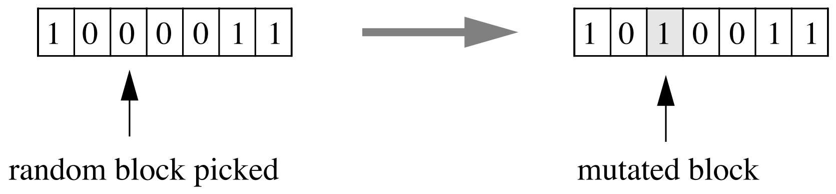


Figure 3 — The Mutation Operator

being generated randomly. Then the population is manipulated in as many iterations as necessary to generate acceptable solutions for the problem or until resources, such as time, have run out.

In each iteration, a *mating pool* is picked from the current population. Members of the current population are picked weighted randomly: the fitter a member, the more chance it is given to enter the mating pool. One may allow the same member of the current population to be picked repeatedly, thus allowing more than one copy of it in the mating pool. Then a new population is generated from the mating pool.

To generate the new population, a pair of strings is picked from the mating pool at a time to form the *parents*. Parents are crossed over using the method described above to generate new offspring. Mutation is allowed to happen occasionally to the offspring so as to allow new patterns of building blocks to appear. This process is repeated until the new population contains as many members as the original one.

Figure 4 shows the data to be manipulated, the control flow (the arrows) and the major GA operators:

(1) initialization — to generate the initial population;

(2) reproduction — to create the mating pool from the current population;

(3) parents selection — to pick parents from the mating pool;

(4) crossover — to generate offspring from parents;

(5) mutation — to change values of the building blocks in the offspring.

It should be emphasized that different GAs may vary (sometimes significantly) in these operators. For example, in order to allow all patterns to be generated, (so that the optimal solution is at least given a chance to emerge without relying on mutation), one may ensure that every possible value for every building block appears in at least one member of the initial population. For example, in a binary coding, one may want to ensure that for every building block both 0 and 1 are present in at least one string.

In the reproduction operation, different weights can be assigned to the strings. For example, instead of using the absolute values of the fitness, the strings may be ranked by their fitness in the population; the higher a string ranks, the greater weight it is given.

In the selection of parents, one may decide to pick the parents weighted randomly instead of just randomly. In crossover, preference may be given to certain cutoff positions. One may also crossover the parents in more than one cutoff point. One may want to apply different mutation rates under different situations (in nature, species mutate more frequently in adverse environments). It is even possible to modify the control flow in such a way that no mating pool is formed: in each iteration, a number of (weak or randomly selected) strings in the population are replaced by new offspring.

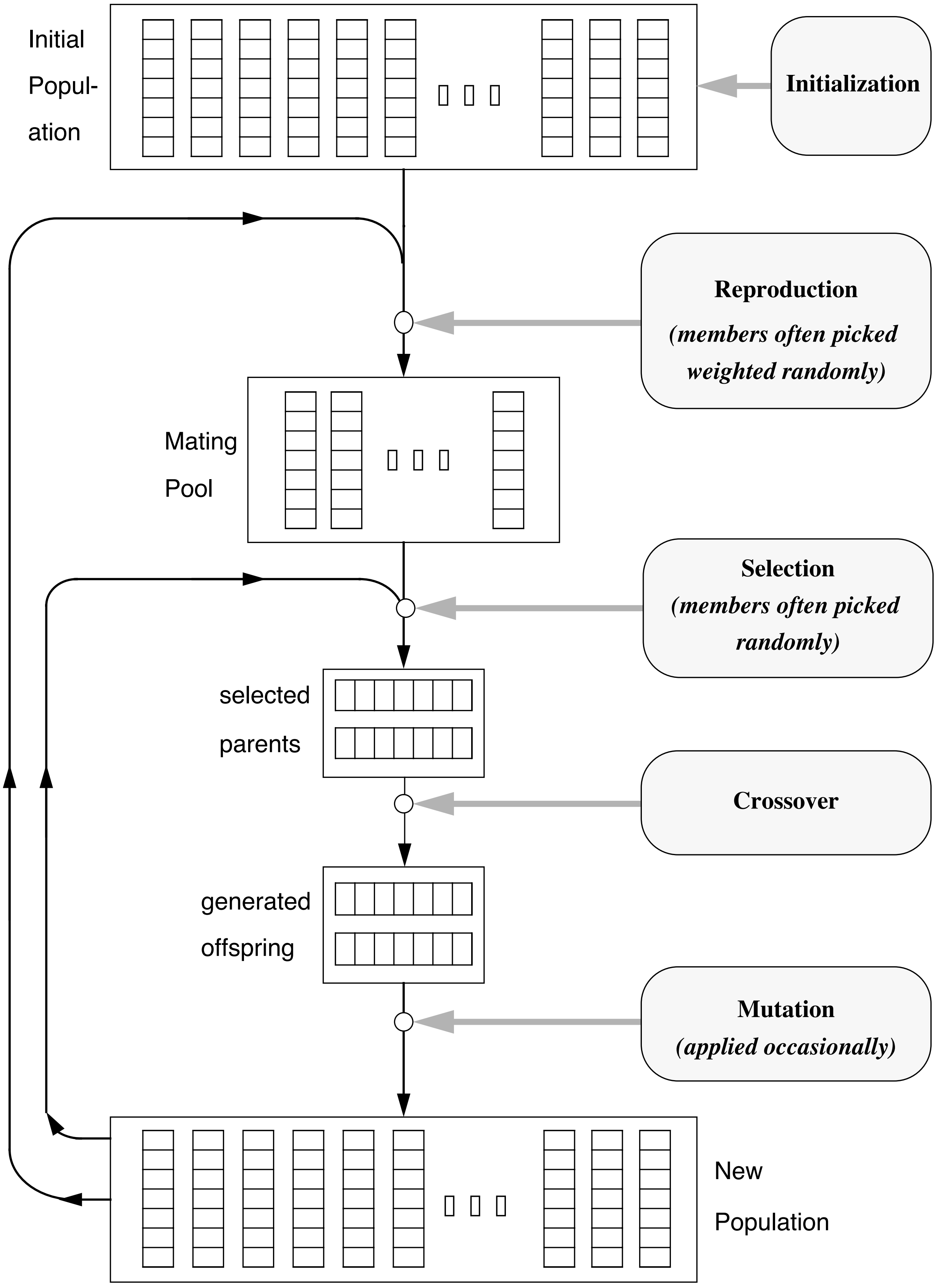
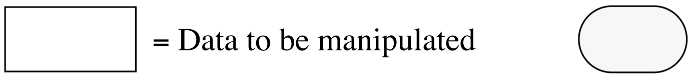


Figure 4 — Overview of the basic control flow and operations in typical GAs



III. A Genetic Algorithm in action

We shall use a simple example to illustrate the GA operators described above. In fact, the problem in this example is so simple that it could have been solved by other methods. Of course, it is the GA operations that we want to show here, not the answer of the problem.

The problem is to find a value for x between 0 and 31 such that $f(x) = 100 + 28x - x^2$ is maximized. The first step is to find a representation which suits GAs. One possibility is to use bit patterns to represent the values that x can take. Since x is between 0 and 31, a 5 bits string should be sufficient. For example, the binary string 10101 represents the value 24. Next, an evaluation function is required to evaluate the fitness of each string. Since this is a maximization problem, the evaluation function can simply be f(x), the function which is to be maximized. This representation and evaluation function satisfy the requirement that the fitness is computed solely from the values of a string's build blocks.

For simplicity, let us assume that a population contains four members only. Let the strings in column (b) of Table 1 be created:

Table 1: Initial Population

(a) string number	(b) strings	(c) dec. numbers represented	(d) $f(x) = 100 + 28x - x^2$	(e) relative weights	(f) accumulated weights
1	01010	10	280	(280/1018) = 28	28
2	01101	13	295	(295/1018) = 29	57
3	11000	24	196	(196/1018) = 19	76
4	10101	21	247	(247/1018) = 24	100
			accum: 1018 average: 254.5	100	

Column (c) of Table 1 shows the decimal values of the binary numbers shown in column (b). Let us assume that strings are selected for the mating pool weighted randomly, where the weights are derived directly from the fitness of the strings. Column (d) shows the fitness of the individual strings. To assign relative weights to the strings, we sum the fitness values (which

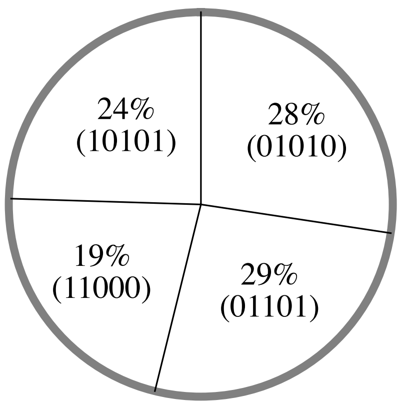


Figure 5 — roulette wheel for selecting strings for the mating pool

gives 1018), and divide each of the fitness values by it. This gives the values in column (e). To form the mating pool, one random number between 0 and 100 is generated at a time. Let us assume that the number 43 is generated. Since 43 is between 28 and 57, the second string is selected (refer to column (f) of Table 1). The effect is equivalent to dividing up a roulette wheel by the weights as it is shown in figure 5, and decide which string to pick by rolling the wheel and checking where the ball stops. Naturally, the first and the second strings have better chance of being selected, as they have bigger shares of the wheel. Let us assume that four strings are put into the mating pool, and the random numbers 43, 27, 89 and 14 are generated. This gives two copies of string 1, and one copy of strings 2 and 4. The mating pool created is shown in Table 2.

Table 2: Example of a Mating Pool

(a) string references	(b) strings	(c) decimal numbers represented
m_1	01101	13
m_2	01010	10
m_3	10101	21
m_4	01010	10

Assume that strings m_3 and m_4 in the mating pool are picked randomly to form parents. Assume further that the randomly picked cutoff point is between building blocks 2 and 3. The

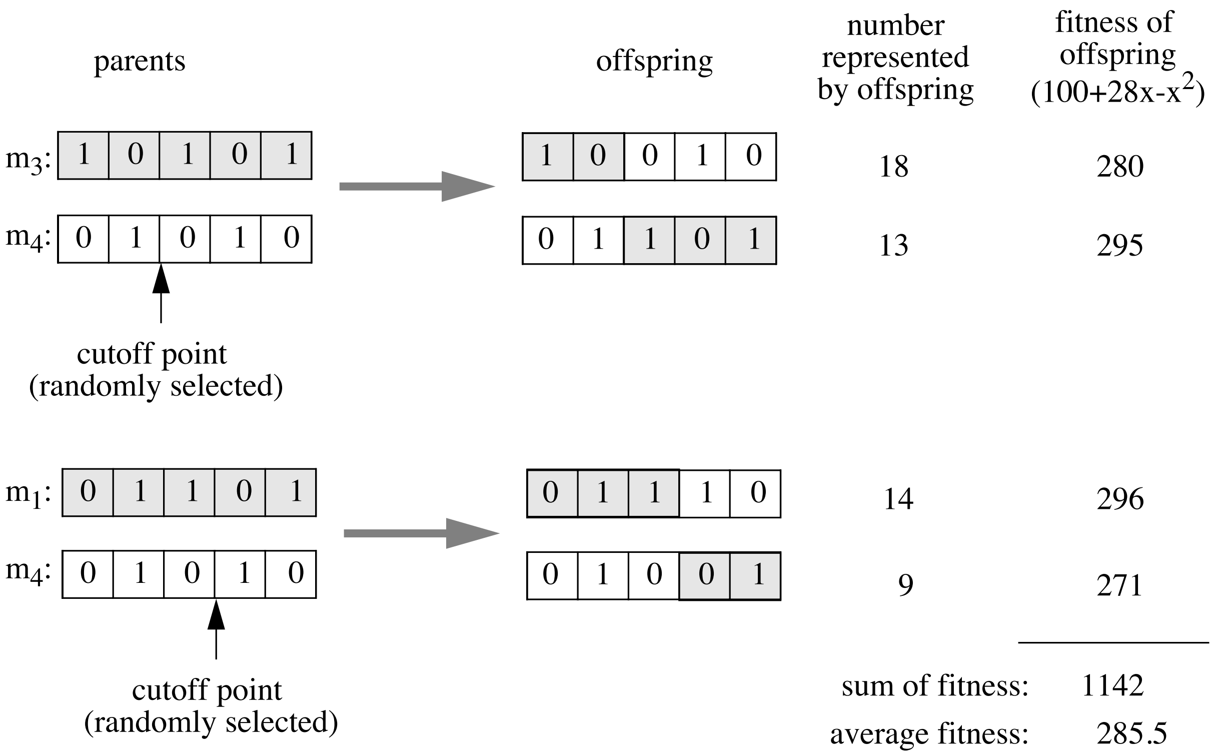


Figure 6 — Example showing the generation of four offspring, which form the new population

result of crossover is shown in the upper part of figure 6. A mutation rate may be set to one in every thousand offspring generated. If, say, the offspring 10010 is selected for mutation, a random bit in it will have its value changed. Example of an outcome is 11010 (value of the second bit changed).

To complete our example of generating the next population, let us assume that stings m_1 and m_4 are picked from the mating pool as parents for generating the next pair of offspring. Let the cutoff point be between building blocks 3 and 4 this time. The result of the crossover is shown in the lower part of figure 6. The four offspring shown in figure 6 form the new population. The whole process of population generation can be repeated as many times as necessary, or until resources have run out.

In figure 6, we have computed the fitness of all the offspring and the sum and average fitness of the population. After the iteration shown in this example, the average fitness of the population is improved from 254.5 (refer to column (d) of Table 1) to 285.5. This in fact shows a typical picture of GAs: it is provable that in GAs, the average fitness of the population tend to grow over iterations. Therefore, the more iterations a GA is allowed to run, the more chance it has in finding fitter strings.

IV. Performance of GAs

The GAs outlined above are very simple to implement. But how well do they work? What have they been applied to? The answer is that GAs have been applied to a wide variety of problems, and more and more success have been reported. GAs have out-performed many conventional methods in terms of robustness, efficiency and effectiveness.

GAs have been applied successfully to optimization problems. For example, a GA has been applied to travelling salesman problems (TSPs) with 500 cities and found better results than conventional methods developed for tackling TSPs. GAs have also been applied to *Quadratic Assignments Problems* (QAPs) (the assignment of *n* values to *n* variables optimizing complex cost functions), and outperformed specialized mathematical methods developed for tackling them. Such results are remarkable because both the TSP and the QAP are very general problems and years of research have been spent on each of the conventional methods developed specifically for tackling them. In contrast, GAs are general methods which are adapted to tackle these problems. More remarkably, GAs require less time to find better solutions than most conventional methods in these problems.

In many expert systems knowledge is encoded in condition-action rules. One important application of GA is in generating such rules. Writing down rules for an expert system is well known to be both laborious and error-prone. Very often, even experts in their fields cannot encode their knowledge as rules. Besides, it is difficult to know whether all the important and relevant rules have been written down. GAs have been used to “cultivate” rules for such systems. This can be done by either using each string to represent a set of rules (in which case the string would normally be quite long) or to have the population as a whole representing a set of rules. Rule sets generated by GAs are called *classifier systems*. Robustness is one of the strength of using GAs to generate classifier systems.

One successful example of applying GAs to generate rules is the Prisoners’ Dilemma Problem. Given a payoff table, the problem is to find a strategy which maximizes a player’s payoff. A strategy is basically a program which responds under various situations. A GA has successfully generated a strategy which behaves very similarly to the best strategy found so far (for readers who are interested, the best strategy found so far is called *tit-for-tac*). Research in using GAs for other applications is abundant, some of which are shown to be more promising than others. Examples of other GA applications in which success have been reported are scheduling, structural optimization, etc. Some GA-based schedulers have outperformed both human and computer schedulers that they are developed to replace, both in schedules generation times and the quality of the schedules generated. In the University of Essex, current research in GAs includes applying GAs to neural network configuration, scheduling and constraint satisfaction. Besides, two graduate students are doing their projects on evolution modelling.

V. Summary

Genetic Algorithms (GAs) are computer algorithms which are inspired by evolution in nature. They have been applied to a large number of optimization problems and rules generation problems. Remarkable success has been achieved.

In order to apply GAs, one needs to be able to encode the candidate solutions by strings of building blocks and evaluate the quality of a string solely by the values that its building blocks take. A GA works by manipulating populations of such strings in search of quality strings. The simplicity, robustness, efficiency and effectiveness of GAs make them promising tools for complex applications. □